CMPT 777 Programming Assignment 3

Sihui Wang 301474102

1. For this question, what we need to do is to annotate the loop invariants, so that it is possible for Dafny to work with loops.

Loop invariant No.1: Range of Loop Variable `i`:

We need to make sure that the invariant holds for every execution of the loop, including the very last one. So, the loop variant for `i` should be:

```
invariant 0 <= i <= a.Length
```

Loop invariant No. 2: The "not found yet" indicator:

The program executes `i := i + 1` and goes to the next iteration of the loop because it hasn't found `v` from `a[0]` to `a[i - 1]` so far. We need to provide this information as the loop invariant:

```
invariant forall k :: 0 <= k < i ==> a[k] != v
```

Given the above 2 loop invariants, the program can be verified by Dafny.

2. For this question, what we need to do is to annotate the loop invariants, so that it is possible for Dafny to work with loops.

Loop invariant for the inner loop:

For the inner loop of `j`, `j` starts at `j := i` and ends with `j := 0`. When `j = 0`, the program will not enter the body of the loop, so the body of the loop is executed for `i` times, each adding 10 to the variable `k`. So, the formula for `k` is: $k = (i - j) \cdot 10$, and we need to provide this as a loop invariant:

```
invariant k == (i - j) * 10
```

Loop invariant for the outer loop:

For the outer loop of `i`, `i` starts at `i := n`. For each execution of the loop body, the value `i * 10` is added to the variable `sum`. So, at the entry point of the loop, `n * 10, … ,(i + 1) * 10` have been added to the variable `sum`. So, the formula for `sum` is: $sum = \frac{(n+i+1)\cdot(n-i)}{2} \times 10$, and we need to provide this as a loop invariant:

```
invariant sum == (n + i + 1) * (n - i) * 5
```

Given the above 2 loop invariants, the program can be verified by Dafny.

3.

Declaring Pre- and Post-conditions:

The question says that the array is non-empty, so we need to provide the pre-condition:

```
requires a.Length > 0
```

For the post-condition part, just "translate" the requirements in the question to Dafny language.

min is less than or equal to all elements in the array:

```
ensures forall k :: 0 <= k < a.Length ==> a[k] >= min
```

min is equal to some element in the array:

```
ensures exists k :: 0 <= k < a.Length && a[k] == min
```

Program Body:

At the beginning, we set `min := a[0]`. Then, we iteratively read from `a[i]` (`1 <= i <= a.Length - 1`) and compare `a[i]` with `min`. If `a[i]` is less than `min`, then we set `min := a[i]` so that `min` is the minimum so far from `a[0]` to `a[i]`. At the end of the program, `min` will be the minimum for the whole array. Since `min` gets it value from some `a[i]`, we are sure that `min` is equal to some element in the array.

Loop invariants:

1) Range of Loop Variable `i`:

We need to make sure that the invariant holds for every execution of the loop, including the very last one. So, the loop variant for `i` should be:

```
invariant 1 <= i <= a.Length
```

2) the "Minimum So Far" Indicator:

When the program executes `i := i + 1` and goes to the next iteration, it is ensured that `min` is less than or equal to all elements from `a[0]` to `a[i - 1]`. It is also ensured that `min` is equal to some element from `a[0]` to `a[i -1]`. We just need to declare these as the loop invariants:

```
invariant forall k :: 0 <= k < i ==> a[k] >= min
invariant exists k :: 0 <= k < i && a[k] == min
```

4.

This question is a simplified version of the example code "`DutchFlag.dfy`".

Data Type:

We need to create a `datatype CoinSide` with two possible values: `Front` and `Back`.

```
datatype CoinSide = Front | Back
```

Predicate Indicating the Orders:

We need to define that `Front` should always be before `Back`. We write this as a `predicate Before`:

```
predicate Before(a: CoinSide, b: CoinSide)
```

```
{
      a == Front || a == b
}
```

Framing[1]:

For the method `SortCoins`, we need to declare that it modifies the input array:

```
method SortCoins(a: array<CoinSide>)
      modifies a
      …
```

Post-Conditions:

We need to declare that `Front` should occur before `Back` using the `predicate Before`:

```
  ensures forall i, j :: 0 <= i < j < a.Length ==> Before(a[i], a[j])
```

We also need to declare that the sorted array is a permutation of the original array:

```
            ensures multiset (a[..]) == multiset (old(a[..]))
```

Program Body:

We set two variables `f` and `b`, `f` starting from `0` and `b` starting from `a.Length`. We need to make sure that all elements before `a[f]` (excluding `a[f]`) are `Front` and all elements after `a[b]` (including `a[b]`) are `Back`. At the end, when `f` and `b` meet (i.e., `f=b`), we ensure that the whole array is sorted.

To do this, when the array is not fully sorted (i.e., `f < b`), we check whether `a[f]` is `Front` or `Back`. If `a[f]` is `Front`, we can increase `f` by `1` and make sure that all elements before `a[f]` are `Front`. If `a[f]` is `Back`, then we can decrease `b` by `1`, swap `a[f]` and `a[b]`, and make sure that all elements after `a[b]` are `Back`. We iteratively repeat this procedure until `f=b`, which can guarantee that all elements are sorted.

```
var f, b := 0, a. Length;
while(f < b)
{
      match a[f]
      case Front =>
            f := f + 1;
      case Back =>
            b := b – 1;
            a[f], a[b] := a[b], a[f];
}
```

Loop Invariants:

1) Range of `f` and `b`:

`f` starts from `f :=0` and `b` starts from `b := a.Length`. In the loop, `f` is increasing and `b` is decreasing until `f = b`. So, the loop invariant is:

```
invariant 0 <= f <= b <= a.Length
```

2) "Partially Sorted" Indicator:

The program can guarantee that all elements before `a[f]` (excluding `a[f]`) are `Front` and all elements after `a[b]` (including `a[b]`) are `Back`. We need to declare these in the loop invariants:

```
invariant forall i :: 0 <= i < f ==> a[i] == Front
invariant forall i :: b <= i < a.Length ==> a[i] == Back
```

3) Permutation Indicator:

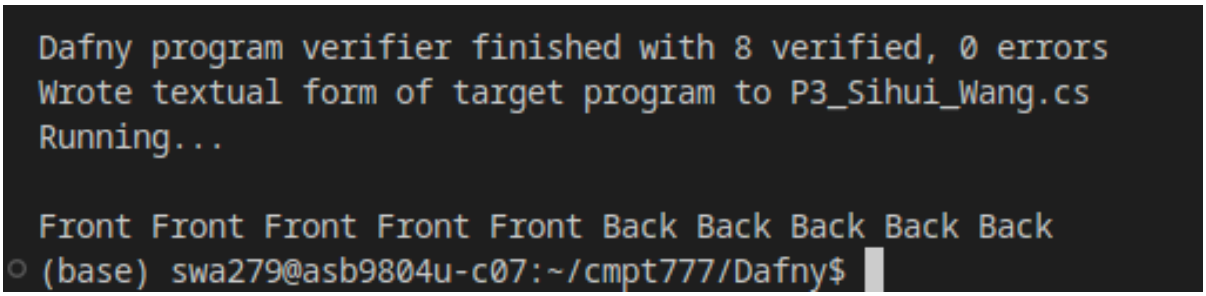
We need to declare that every update of the array is a permutation:

```
invariant multiset (a[..]) == multiset (old(a[..]))
```

Given the above loop invariants, the program can be verified by Dafny.

Main Method:

In the `Main()` method, I also give a concrete example for question 4. The input is an unsorted `CoinSide` array of size 10, and the output is the sorted array.



Reference:

[1] https://ece.uwaterloo.ca/~agurfink/stqam.w20/rise4fun-Dafny/